FUEL/QOPQDP replacement

James C. Osborn & Xiao-Yong Jin

ALCF

USQCD Software call
Jan. 21, 2016

FUEL/QOPQDP

- QOPQDP (and QDP/QLA)
  - Started in 2001
  - Well optimized for few-core, short-vector machines
  - Somewhat complicated set of Perl code generators creating large libraries
- FUEL
  - Wraps QOPQDP/QDP in high level scripting language (Lua)
  - Very convenient for writing new code, experimenting with algorithms, etc.
  - Loose efficiency and flexibility of writing to lower level
- Needs significant overhaul to make efficient use of future architectures while keeping high level, easy to use, scripting (-like) interface

FUEL/QOPQDP redesign initial plans

- Make flexible code generator
  - Write high-level expressions
  - Generate efficient low-level C code
  - Could make similar high-level interface available at compile-time and run-time
- Wanted true code-generator
  - Full control over generated code and high level constructs
  - Able to analyze, transform and generate code in a natural, powerful, high-level language
  - C++ templates are Turing-complete (technically have full control), but not at all natural or high-level
- Initially considering constructing code generator in Lua for lack of a better alternative, until I discovered...

Nim (nim-lang.org)

- Modern language started in 2008
- Designed to be “efficient, expressive, and elegant”
- Borrows heavily from: Modula 3, Delphi, Ada, C++, Python, Lisp, Oberon
- Statically typed, but has extensive type-inference, so feels like dynamically-typed scripting language
- Efficient garbage collection (optional)
- Extensive meta-programming support (nearly full language available at compile time)
- Final output is C or C++ (or JS) code
  - GPU (OpenCL) on roadmap, but probably long ways off
- LLVM-IR backend recently contributed (still in development)

Nim (nim-lang.org)

- Easily interfaces with existing C/C++ code
- Allows inserting C/C++ code directly in output
- Openly available on github (MIT license)
- Started by single person (Andreas Rumpf) who is main developer
- 10 contributors with 50+ commits in past year, 79 total in past year
- Active forum on website with very knowledgeable contributors
- Under active development
  - Current version 0.13 (Jan. 18)
  - Language still evolving, but mostly stable
  - Many open issues (646), most minor, but several (22) high priority
- A few small companies using it (game, web)

Generic and meta-programming features

C++	Nim
preprocessor macros	templates: inline code substitutions also allows overloading, completely hygenic (if desired)
templates	generics: applies to type definitions, procedures, templates and macros also allows typeclasses, concepts
???	macros: similar to lisp: syntax tree of arguments passed to macro at compile time to allow arbitrary manipulation

New lattice framework

- Writing new lattice expression framework in Nim
- Using recently developed QLL for layout/communications framework
  - Staggered CG ~23% on BG/Q
- Can get hand-written Nim code to nearly match performance of hand-written C code
- Working on generating efficient code from expressions (much easier on x86 since compilers are newer)
- Have done many tests to understand language and capabilities
- Just recently started putting pieces together into coherent package
- Trying to get simple examples up and running to get it usable
- Will focus on adding features, optimization and refining syntax as we go

QEX: QCD (or Quantum) Expressions

```
import qex
import qcdTypes

qexInit()
var lat = [4,4,4,4]
var lo = newLayout(lat)
var v1 = lo.ColorVector()
var v2 = lo.ColorVector()
var m1 = lo.ColorMatrix()
threads:
  m1 := 1
  v1 := 2
  v2 := m1 * v1
  shift(v1, dir=3, len=1, v2)  # len=+1: from forward
  single:
    if myRank==0:
      echo v2[0][0]  # vector “site” 0, color 0
qexFinalize()
```

QEX/Nim examples

- threads: implementation

```
template threads*(body:untyped):untyped =
  let tidOld = tid
  let nidOld = nid
  proc tproc =
    {.emit:"#pragma omp parallel".}
    block:
      setupForeignThreadGc()
      tid = ompGetThreadNum()
      nid = ompGetNumThreads()
      body
  tproc()
  tid = tidOld
  nid = nidOld
```

QEX/Nim examples

```
var v3 = lo.ColorVector()
template S0(x:v3.type):expr =
  shift(v3, dir=0, len=1, x)
  v3

threads:
  for s in v1.all:
    var aa:array[VLEN,float32]
    for i in 0..<VLEN:
      aa[i] = (((x[0][i]*10+x[1][i])*10+x[2][i])
                 *10+x[3][i]).float32
    v1[s][0].re := aa

  v2 := m1 * v1.S0
```

QEX tensors

- General site-wise tensor support in development:

```
type
  Color = range[1..3]
  Spin = range[1..4]
  HalfSpin = range[1..2]
  CVec = nameTensor(complex, Color)
  SCVec = nameTensor(complex, Spin, Color)
  CMat = nameTensor(complex, Color, Color)
  SCMat = nameTensor(complex, Spin, Spin, Color, Color)
var
  d1,d2: SCVec
  p1: SCMat

d1[s,c] <- (if s==1 and c==1: 1.0 else: 0.0)
d2[mu,a] <- p1[mu,nu,a,b] * d1[nu,b]
t <- p1[mu,mu,a,a]
```

QEX/Nim configuration & compilation

- Nim automatically keeps track of dependencies (import's) and will compile and link all sources needed to produce executable, no Makefile necessary!

```
nim c myProject1.nim
nim c myproject2.nim
…
```

- Setup configuration file:

```
cc = gcc
gcc.exe = mpicc
gcc.linkerexe = mpicc
gcc.options.always = "-Wall -std=gnu99"
gcc.options.speed = "-O3 -march=native"
gcc.options.debug = "-g3 -O0"
```

- C wrappers can automatically include headers/libraries; example from qmp.nim:

```
when not defined(qmpDir):
  const homeDir = getHomeDir()
  const qmpDir = homeDir & "lqcd/install/qmp"
{. passC: "-I" & qmpDir & "/include" .}
{. passL: "-L" & qmpDir & "/lib -lqmp" .}
{. pragma: qmp, importc, header:"qmp.h" .}
proc QMP_get_node_number*():cint {.qmp.}
```

QEX/Nim scripting

- Having scripting interface to application provides:
  - Flexible, procedural, interface to set up parameters
  - Avoids recompiling for simple changes in workflow or need to maintain Makefiles for new codes
  - Enables rapid testing and development by providing high level interface to routines

- Nim provides most of this, except for the actual compiling (so far compile times are < few seconds)
- Could plug in Lua
- Nim provides its own scripting interface (Nimscript)
  - Used in compiler for compile-time evaluation
  - Available to plug in to application and can interface with rest of application

QEX plans

- Short term
  - Finish shifts, parallel transport functions
  - Add I/O (QIO)
  - Integrate site-wise tensor expressions
  - At this point it should be usable and reasonably efficient for most applications
  - Add ability to explicitly create CUDA kernels
- Longer term
  - Implement optimized field-wise expressions including shifts
  - Make CUDA code generation automatic from threads: region